From Causal Consistency to Sequential Consistency in Shared Memory Systems

Michel Raynal, André Schiper

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From Causal Consistency to Sequential Consistency in Shared Memory Systems

M. Raynal, A. Schiper

N° 2557

Mai 1995

PROGRAMME 1

ISSN 0249-6399

From Causal Consistency to Sequential Consistency in Shared Memory Systems

M. Raynal*, A. Schiper**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n°2557 — Mai 1995 — 27 pages

Abstract: Sequential consistency and causal consistency constitute two of the main consistency criteria used to define the semantics of accesses in the shared memory model. An execution is sequentially consistent if all processes can agree on a same legal sequential history of all the accesses; if processes perceive distinct legal sequential histories of all the accesses, the execution is only causally consistent (legality means that a read does not get an overwritten value).

This paper studies synchronization constraints that, when obeyed by operations of a given causally consistent execution, make it sequentially consistent. More precisely, the paper introduces the MSC synchronization (mixed synchronization constraint) which generalizes (1) the known DRF (data race free) and CWF (concurrent write free) synchronizations and (2) a new one called CRF (concurrent read free). The MSC synchronization allows for concurrent conflicting operations on a same object, while ensuring sequential consistency; this is particularly interesting in the the context of distributed systems (where objects are possibly replicated) to cope with partition failures: conflicting operations in two distinct partitions do not necessarily block processes that issue them (as it is the case of quorum based protocols).

Key-words: Causal consistency, sequential consistency, shared memory, synchronization constraint.

(Résumé : tsvp)

*IRISA, Campus de Beaulieu, 35042 Rennes-Cédex, raynal@irisa.fr.
**EPFL, Dept d'Informatique, 1015 Lausanne (Switzerland) schiper@di.epfl.ch.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

De la cohérence causale à la cohérence séquentielle dans les mémoires partagées

Résumé : La cohérence causale et la cohérence séquentielle constituent deux des principaux critères utilisés pour définir la sémantique des accès dans le modèle "mémoire partagée". Ce rapport étudie les liens entre ces deux types de cohérence et montre qu'une exécution causalement cohérente dont les opérations de lecture et d'écriture respectent certaines contraintes de synchronisation est en fait séquentiellement cohérente (une exécution est séquentiellement cohérente si tous les processus voient la même histoire légale des accès à la mémoire). Une contrainte de synchronisation est particulièrement étudiée; son intérêt pour résister au partitionnement dans le contexte des systèmes répartis est mis en valeur.

Mots-clé : contrainte de synchronisation, cohérence causale, cohérence séquentielle, mémoire partagée.

1 Introduction

For several years the shared memory model has become a pervasive concept in parallel and distributed systems. This is due to the universality of the model: processes distributed over a network and interacting through shared objects (objects distributed over the network, and possibly replicated), fit for example perfectly into this model. Moreover, the shared memory model[1] is the adequate framework for defining consistency criteria: a consistency criterion defines the value returned by every read operation invoked by a process on some object (or some variable). It is important to stress that the definition of a consistency criterion must be independent of the possible existence of multiples copies of objects, and must not rely on a particular protocol implementing the criterion; it must be based on a formal model and be as general as possible to make designers capable to study properties of consistency criteria, and to produce results not bound to particular implementations. As for abstract data types, such a classical approach distinguishes clearly the semantics offered to users from its particular implementations. Several authors have correctly claimed that a memory consistency criterion is a contract between the memory system and application programs [21].

Three main consistency criteria have been proposed in the literature: atomic consistency [19] (also called linearizability [14]), sequential consistency [17] and causal consistency [3]. In all three cases a read operation returns the *last* value assigned to the variable (or written into the object). The three consistency criteria differ however in the definition of the *last* write operation. Atomic consistency is the more restrictive of the three consistency criteria: it requires that all the processes agree on a total order including all the read/write operations that they have issued, and this total order has to respect real time (i.e. if op_1 precedes op_2 in real time, then all the processes have to agree that op_1 has occurred before op_2). With sequential consistency, the processes have also to agree on a total order of their read/write operations, but this total order does not have to respect their real time order. With causal consistency, processes can disagree on the way they totally order concurrent write operations.

Causal consistency is included in sequential consistency (i.e. an execution that satisfies sequential consistency also satisfies causal consistency) and sequential consistency is included in atomic consistency (i.e. an execution that satisfies atomic consistency also satisfies sequential consistency). If some relationships between atomic and

[1] A (logically) shared memory includes simultaneously all the objects with all their implicit mutual and causal dependencies.

sequential consistencies are well understood (e.g. [5] compares their respective powers), a unifying framework, that would allow for a better understanding of the links between sequential consistency and causal consistency criteria, is still missing. This is precisely the purpose of this paper, which shows that sequential consistency can be obtained from causal consistency by adding some appropriate synchronization constraints (a synchronization constraint orders some pair of operations). This is particularly interesting from methodological and implementation points of view as it means that a family of protocols implementing sequential consistency can be seen as consisting of two independent layers: a basic layer implementing causal consistency and, on top of it, another layer enforcing the chosen synchronization constraints (an interesting consequence of the approach is that only this second layer has to be changed when we want to replace a set of synchronization constraints by another one).

Our work can be seen as a continuation of the work started by Ahamad *et al.* [4]. These authors consider however only two types of synchronization constraints: *Data Race Free* (DRF) synchronization, and *Concurrent Write Free* (CWF) synchronization. We generalize their work in two directions. First we distinguish between two classes of synchronization constraints: (1) the *per object* synchronization constraints, which synchronizes operations on each object independently, and (2) the *inter-object* synchronization constraints, which synchronizes operations on distinct objects. The DRF synchronization fits into the per object synchronization class, and the CWF synchronization fits into the inter-object synchronization class. The per object synchronization class is particularly interesting, as it provides the *locality* property introduced by Herlihy and Wing [14]. The second, and most important contribution of the paper, is the introduction of two new synchronization constraints: the *Concurrent Read Free* (CRF) synchronization, and the *Mixed Synchronization Constraint* (MSC) which combines the DRF, CWF and CRF synchronizations. The MSC synchronization has the nice property to allow conflicting operations on the same object to proceed concurrently. This is particularly interesting in the context of distributed systems where objects are possibly replicated, to cope with partition failures. If sequential consistency is obtained by implementing the MSC synchronization on top of a causally consistent distributed shared memory, then conflicting operations issued from two distinct partitions do not necessarily block processes that issued them (when sequential consistency is ensured by quorum based protocols such blocking always occurs).

The paper is structured as follows. Section 2 formally defines causal and sequential consistency. Section 3 introduces the two classes of synchronization constraints

(per object synchronization and inter-object synchronization), and defines the DRF, CWF and CRF synchronization constraints. The mixed synchronization constraint MSC is introduced in Section 4. This Section also contains the proof of the main Theorem: GSC allows to transform causal consistency into sequential consistency. Section 5 addresses practical aspects of the MSC synchronization in the context of distributed systems (implementation of the constraint and partition failures). Finally Section 6 discusses other consistency criteria from which sequential consistency can also be obtained.

2 Shared Memory Model

2.1 Notations

We consider a finite set of sequential processes $P_1, \ldots, P_n$ that interact via a finite set X of shared objects. Each object $x \in X$ can be accessed by read and write operations. A write into an object defines a new value for the object; a read allows to obtain a value of the object. A write of value v into object x by process P_i is denoted $w_i(x)v$; similarly a read of x by process P_j is denoted $r_j(x)v$ where v is the value returned by the read operation; *op* will denote either r (read) or w (write). For simplicity, we assume all values written into an object x are distinct. Moreover, the parameters of an operation are omitted when they are not important. Each object has an initial value; it is assumed that this value has been assigned by an initial fictitious write operation.

2.2 Histories

The *local history* $\widehat{h}_i$ of P_i is the sequence of operations issued by P_i. If $op1$ and $op2$ are issued by P_i and $op1$ is issued first, then we say $op1$ precedes $op2$ in P_i's process-order, which is noted $op1 \rightarrow_i op2$. Let h_i denote the set of operations executed by P_i; the local history $\widehat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

An *execution history* (or simply a history) $\widehat{H}$ of a shared memory system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that[2] :

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ if :

[2] Section 6.5 briefly compares definition of histories in the shared memory model and in the message-passing model.

i) $\exists\ P_i\ : op1 \rightarrow_i op2$ (in that case, $\rightarrow_H$ is called *process-order* relation),

or *ii)* $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (in that case $\rightarrow_H$ is called *read-form* relation),

or *iii)* $\exists op3\ : op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$.

A history $\widehat{H}$ is *sequential* if $\rightarrow_H$ is a total order relation.

A read operation $r(x)v$ is *legal* if: $\exists\ w(x)v\ : w(x)v \rightarrow_H r(x)v$ and $\nexists\ op(x)u\ : w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v$. A history $\widehat{H}$ is legal if all its read operations are legal[3].

Two histories are *equivalent* if (1) they are defined on the same set of operations, (2) they have the same process-order relation, and (3) they have the same read-from relation.

Two operations $op1$ and $op2$ are *concurrent* in $\widehat{H}$ if we have neither $op1 \rightarrow_H op2$ nor $op2 \rightarrow_H op1$.

2.3 Sequential Consistency

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [17]. Such a system is sequentially consistent with respect to a multiprocess program, if "*the result of any execution is the same as if (1) the operations of all the processors where executed in some sequential order, and (2) the operations of each individual processor appear in this sequence in the order specified by its program*".

This informal definition states that the execution of a program is sequentially consistent if it is equivalent to a sequential execution[4]. More formally, we define sequential consistency in the following way.

[3] The usual definition of legality [3, 4] eliminates only the possibility of an intervening write ($w(x)u$) between the writing of some value ($w(x)v$) and a reading of the same value ($r(x)v$). Our definition of legality allows for a simpler definition of causal consistency (see Section 2.4).

[4] In his definition, Lamport assumes that the *process-order* relation defined by the program (see point *(2)* of the definition) is maintained in the equivalent sequential execution, but not necessarily in the execution itself. As we do not consider programs but only executions, we implicitly assume that the *process-order* relation displayed by the execution histories are the ones specified by the programs which gave rise to these execution histories.

Definition. Sequential consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *sequentially consistent* if there exists a legal sequential history $\widehat{S}$ equivalent to $\widehat{H}$. In other words, $\widehat{H}$ admits a linear extension[5] $\widehat{S}$ in which all reads are legal. □

As an example let us consider the history $\widehat{H_1}$ (Figure 1)[6]. Each process P_i, (i=1,2), has issued three operations on the shared objects x and y. The write operations $w_1(x)0$ and $w_2(x)1$ are concurrent. It is easy to see that $\widehat{H_1}$ is sequentially consistent by building a legal sequential history including first the operations issued by P_1 and then the ones issued by P_2. It is also easy to see that the history $\widehat{H_2}$ (Figure 2) is not sequentially consistent, as no equivalent legal sequential history can be built.

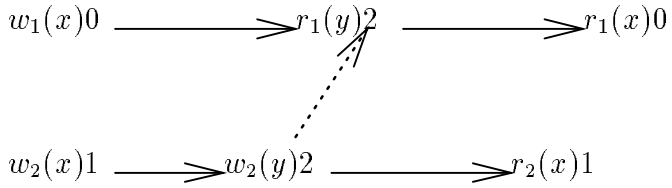

Figure 1: A sequentially consistent history $\widehat{H_1}$

Various cache-based protocols implementing sequential consistency have been proposed in the context of parallel machines [1, 5, 20]. The protocols presented in [1, 20] allow several read operations and one write operation to concurrently access a same variable (reading of cached values and writing into the main memory) but do not allow concurrent write operations on a same variable. One of the protocols (called *fast write*) presented in [5] allows write operations on a same variable to proceed concurrently. However, these protocols do not assume an underlying causally consistent memory, and thus could not identify the two layers approach and the mixed synchronization constraint given in the paper.

In the context of distributed systems, where each object is supported by several permanent copies, non cached-based protocols implementing sequential consistency

[5] A linear extension of a partial order is a topological sort of this partial order, so it maintains the order of all ordered pairs of the partial order.

[6] In all figures, only the edges that are not due to transitivity are indicated (transitivity edges come from *process-order* and *read-from* relations). Moreover, (intra-process) *process-order* edges are denoted by continuous arrows and (inter-process) *read-from* edges by dotted arrows.

have been proposed. Usually these protocols use votes [24] or quorums [13] mechanisms and, consequently, implement actually atomic consistency which is stronger than sequential consistency. Section 5.1 discusses some of these protocols.

2.4 Causal Consistency

Causal consistency, introduced by Ahamad *et al.* in 1991 [3], defines a consistency criterion weaker than sequential consistency. Causal consistency allows for a wait-free implementation of read and write operations in a distributed environment, i.e. causal consistency allows for cheap read/write operations (see [3, 4] for protocols implementing causal consistency).

With sequential consistency, all processes agree on a same legal sequential history $\widehat{S}$. The agreement defined by causal consistency is weaker. Given a history $\widehat{H}$, it is not required that two processes P_i and P_j agree on the same ordering for the write operations which are not ordered in $\widehat{H}$. The reads are however required to be legal.

Definition. Causal consistency. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. $\widehat{H}$ is *causally consistent* if all its read operations are legal. □

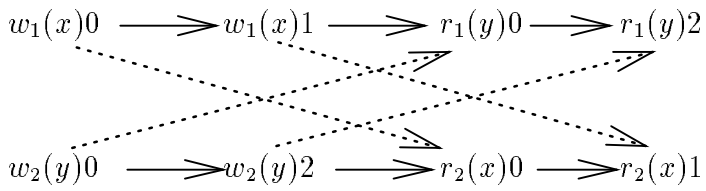

Figure 2: A causally consistent history $\widehat{H_2}$

In a causally consistent history, all processes see the same partial order on operations but, as processes are sequential, each of them might see a different linear extension of this partial order. An alternative definition of causal consistency is the following one [4]. Let $\widehat{H_i}$ be the sub-history of $\widehat{H}$ from which all read operations not in the local history h_i of P_i have been removed[7]. History $\widehat{H}$ is causally consistent if, for each process P_i, there is a legal sequential history $\widehat{S_i}$ equivalent to $\widehat{H_i}$.

[7] More formally, $\widehat{H_i}$ is the sub-relation of $\widehat{H}$ induced by the set of all the writes of H and all the reads issued by P_i.

So, in a causally consistent history, no read operation of a process P_i can get a value that, from his point of view, has been overwritten by a more "recent" write. As an example consider history $\widehat{H_2}$ (Figure 2). This history is causally consistent as all its read operations are legal. The history $\widehat{H_3}$ (Figure 3) is not causally consistent as the read operation $r_3(x)1$ issued by P_3 is not legal: $w_1(x)1 \rightarrow_H r_3(x)2 \rightarrow_H r_3(x)1$. When considering the alternative definition, when P_3 has issued its first read operation on x (namely $r(x)2$), it has got the value 2, and consequently for this process, the value 1 of x has logically been overwritten.

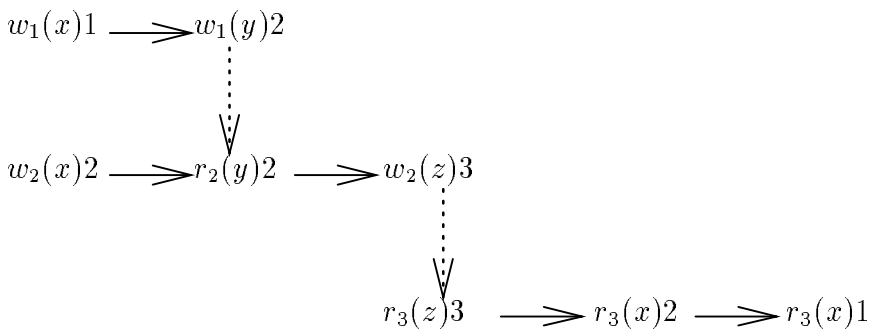

Figure 3: A non causally (but PRAM) consistent history $\widehat{H_3}$

3 Basic Synchronization Constraints

3.1 Adding Synchronization Constraints

As mentioned in the introduction, this paper shows that a causally consistent history whose operations respect some synchronization constraints is a sequentially consistent history $\widehat{H}$: these synchronization constraints ascertain the existence of a legal and sequential history $\widehat{S}$ equivalent to $\widehat{H}$. From an implementation point of view this means that a protocol implementing sequential consistency can be seen as consisting of two independent layers: (1) a first layer implementing causal consistency (i.e. basically ensuring the legality of reads), and (2) a second one enforcing the synchronization constraints.

A synchronization constraint orders some pairs of operations. Let $op(x)$ and $op'(y)$ be such a pair. Two classes of constraints can be defined, depending whether or not x and y are the same object.

1. *Per object synchronization.* In this case the synchronization constraint applies to each object x independently of the others. Two operations $op(x)$ and $op'(y)$, such that x and y are distinct, are never synchronized. This type of synchronization is particularly interesting as it provides the *locality* property introduced in [14].

2. *Inter-object synchronization.* In this case the synchronization constraint orders some pairs of operations $op(x)$ and $op'(y)$ on distinct objects x and y. This type of synchronization is more general (as it includes the *per object* synchronization as a special case). Moreover, as op and op' can each be a read or a write operation, several subclasses of constraints can be envisaged.

We consider below the *data race free* synchronization of the *per object* class, and two types of the *inter-object* class: the *concurrent write free* synchronization, and the *concurrent read free* synchronization. These three basic synchronization constraints can be combined to define what we call the *mixed synchronization constraint.* We show that this mixed synchronization, when added to a protocol implementing causal consistency, leads to sequentially consistent histories.

The synchronization constraints will be defined using the ORD predicate. Two operations satisfy this predicate if they are not concurrent. More formally, let $\widehat{H} = (H, \rightarrow_H)$ be a history, and op, op' be two distinct operations of H:

$$ORD\ (op, op') \stackrel{\text{def}}{=} (op \rightarrow_H op')\ or\ (op' \rightarrow_H op)$$

3.2 Synchronization Tags

Each read or write operation of a history $\widehat{H}$ is associated with one and only one synchronization tag. The possible tags are *DRF (data race free), CWF (concurrent write free), CRF (data read free).* A read operation can be tagged *DRF* or *CRF*; a write operation can be tagged *DRF* or *CWF*. Operation op tagged T, and accessing object x, will be noted $op_T(x)$.

Informally, all operations endowed with the same tag T obey the same synchronization constraint (which is called T). Operations with distinct tags are not synchronized. The next two subsections define synchronization constraints based on tagged operations.

3.3 Per Object Synchronization

As indicated previously, the per object synchronization constraint is defined for each object independently of the others: it consists in ordering conflicting operations on each object x (two operations on an object x are conflicting iff one of them is a write). In other words, operations on each object obey the readers-writers synchronization. This constraint is usually called DRF *(data race free)*.

DRF synchronization. The DRF synchronization orders conflicting operations tagged *DRF*. Let $\widehat{H} = (H, \rightarrow_H)$ be a history and x an object. For any two distinct operations $op_{DRF}(x)$ and $op'_{DRF}(x)$, the DRF synchronization ensures that, if at least one of these operations is a write, then $ORD\ (op_{DRF}(x), op'_{DRF}(x))$ holds. □

It has been proved in [4] that a causally consistent history $\widehat{H}$ in which (using our terminology) all operations are tagged *DRF* is sequentially consistent. The reader might consider this result as not very surprizing. This result is obtained here as a special case of a more general result (see Section 4.3). The general result is indeed more interesting than this special case.

3.4 Inter-Object Synchronization

The DRF synchronization does not order operations on distinct objects x and y. We introduce two synchronization constraints that order operations on distinct objects: one called CWF *(concurrent write free)* applies only to write operations, the other, called CRF *(concurrent read free)*, applies only to read operations.

In both definitions hereafter we consider a history $\widehat{H} = (H, \rightarrow_H)$ and a pair of (not necessarily distinct) objects x and y.

CWF synchronization. This synchronization orders write operations tagged *CWF*. For any two distinct write operations, $w_{CWF}(x)$ and $w_{CWF}(y)$, the CWF synchronization ensures $ORD\ (w_{CWF}(x), w_{CWF}(y))$.

CRF synchronization. This synchronization orders read operations tagged *CRF*. For any two distinct read operations, $r_{CRF}(x)$ and $r_{CRF}(y)$, the CRF synchronization ensures $ORD\ (r_{CRF}(x), r_{CRF}(y))$. □

It has been shown in [4] that a causally consistent history $\widehat{H}$, in which all write operations are tagged *CWF*, is sequentially consistent. This result will also be proved in Section 4.3 as a special case of our more general result. Notice that the CRF

synchronization (with all read operations tagged *CRF*) is not sufficient to ensure sequential consistency out of causal consistency. The CRF synchronization, together with the DRF synchronization for write operations, leads however to this result. This is included in our mixed synchronization constraint.

4 A Mixed Synchronization Constraint

4.1 The Mixed Constraint

The synchronization constraints DRF, CWF and CRF introduced so far can be combined to define a mixed synchronization constraint, noted MSC *(mixed synchronization constraint)*. Section 5.1 discusses its practical use. MSC is a generalization of the previous synchronizations in the sense that it allows for combinations of synchronization constraints (either CWF with DRF, or CRF with DRF). In other words, MSC does not require that the same synchronization tag be attached to every operation of a history. We distinguish between two MSC constraints, MSC_W and MSC_R:

- A history $\widehat{H}$ satisfies the MSC_W constraint if and only if (1) all its operations are tagged either *CWF* or *DRF*, and (2) the *CWF* tagged operations obey the CWF synchronization, and the *DRF* tagged operations obey the DRF synchronization.
  Note that in this case all read operations are necessarily tagged *DRF*, while write operations are tagged *DRF* or *CWF*.

- A history $\widehat{H}$ satisfies the MSC_R constraint if and only if (1) all its operations are tagged either *CRF* or *DRF*, and (2) the *CRF* tagged operations obey the CRF synchronization, and the *DRF* tagged operations obey the DRF synchronization.
  Note that in this case all write operations are necessarily tagged *DRF*, while read operations are tagged *DRF* or *CRF*.

We say that the MSC synchronization is satisfied by a history $\widehat{H}$ if and only if $\widehat{H}$ satisfies either MSC_W or MSC_R. Notice that, in a history $\widehat{H}$ that satisfies the MSC constraint, the tags *CWF* and *CRF* are incompatible. If $w(x)$ is tagged *CWF* in $\widehat{H}$ then all read operations of $\widehat{H}$ are tagged *DRF* (constraint MSC_W). Similarly if a read is tagged *CRF* then all write operations are tagged *DRF* (constraint MSC_R).

In order to understand the MSC_W and MSC_R synchronizations, it is important to understand that they do not require conflicting operations on a given object x be

ordered. Consider a history $\widehat{H}$ and the two cases of conflicting operations (namely, read/write conflict and write/write conflict). Consider first read/write conflicting operations. If $\widehat{H}$ satisfies the MSC_W constraint, and if a write operation w on some object x is tagged *CWF*, whereas a read operation r on x is tagged *DRF*, then these two operations are not ordered by the MSC_W synchronization constraint: hence $w_{CWF}(x)$ and $r_{DRF}(x)$ are not synchronized. Consider now the same conflict with $\widehat{H}$ satisfying the MSC_R constraint: if a write operation w on x is tagged *DRF*, whereas a read operation r on x is tagged *CRF*, then these two operations are not ordered by MSC_R: hence $w_{DRF}(x)$ and $r_{CRF}(x)$ are not synchronized.

The same result holds for write/write conflicts. Suppose the MSC_W constraint is satisfied by $\widehat{H}$; if a write operation w on some object x is tagged *DRF* whereas another write operation w' on x is tagged *CWF*, then these operations are not ordered by MSC_W: hence $w_{DRF}(x)$ and $w'_{CWF}(x)$ are not synchronized, i.e. concurrent conflicting writes are allowed!

Remark. To understand that the DRF and the CWF constraints mentioned in [4] are special cases of the MSC constraint, consider the following explanation. First, the DRF synchronization constraint in [4] is obviously a special case of either the MSC_W or of the MSC_R synchronization constraint, in which all the operations are implicitly tagged *DRF*. Second, the CWF synchronization constraint in [4] is a special case of the MSC_W synchronization constraint in which all the write operations are implicitly tagged *CWF* and obey CWF synchronization, and all the read operations are implicitly tagged *DRF* and obey DRF synchronization (in that case the DRF synchronization is actually a *nil* synchronization as it applies only to read operations of $\widehat{H}$ which are never conflicting!).

4.2 Deterministic Read Operations

Section 4.3 proves our main result, namely: a causally consistent history $\widehat{H}$, that satisfies either MSC_W or MSC_R, is sequentially consistent. Because MSC_W allows for concurrent writes on the same object, MSC_W requires an additional deterministic *read rule*, in order for our main result to hold. This rule defines which value has to be returned by a $r(x)$ operation in case the read operation is aware of two concurrent write operations.

Deterministic read rule. Consider $\widehat{H}$ a causally consistent history obeying the MSC_W synchronization, an object x, and two concurrent writes $w_{DRF}(x)u$ and $w_{CWF}(x)v$. Let $r(x)$ be a read operation such that both $r(x)u$ and $r(x)v$

are legal[8]. Then the read operation returns the value written by $w_{CWF}(x)$, i.e. $r(x)$ returns v. □

4.3 The MSC Theorem

We prove in this Section our main result relating causal consistency to sequential consistency. We prove the results for MSC_W and for MSC_R together.

Theorem 4.1 *Let $\widehat{H} = (H, \rightarrow_H)$ be a causally consistent history such that (1) either MSC_W or MSC_R is satisfied, and (2) in case of MSC_W each read operation $r(x)$ follows the read rule (Sect. 4.2). Then $\widehat{H}$ is sequentially consistent.*

4.3.1 Preliminary Definitions

In order to prove the Theorem 4.1, we introduce the Lemma 4.2. This Lemma is based on two additional relations on the operations of a history $\widehat{H}$: a *logical write-write precedence* relation (denoted $\rightarrow_w$) and a *logical read-write precedence* relation (denoted $\rightarrow_r$).

Logical write-write precedence. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. The write-write precedence relation $\rightarrow_w$ is defined on pairs of write operations, on a same object x, that are concurrent in $\widehat{H}$. By definition, this can happen only if one of them is tagged *DRF* while the other is tagged *CWF* (two writes on a same object, both tagged either *DRF* or *CWF*, are ordered in $\widehat{H}$). The logical write-write precedence relation states that the write tagged *DRF* is logically *before* the write tagged *CWF*.

Definition: Let $w_{CWF}(x)$ and $w_{DRF}(x)$ be two concurrent write operations in $\widehat{H}$. Then we have: $w_{DRF}(x) \rightarrow_w w_{CWF}(x)$. (Moreover $\rightarrow_w$ holds only in these cases.)[9]

Logical read-write precedence. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. The logical read-write precedence relation $\rightarrow_r$ is defined on pairs of read and write operations for each object x.

[8] This means the read is aware of both writes and there is no intervening operation $op(x)a$, with $a \neq u$ and $a \neq v$, in between $w_{CWF}(x)v$ and $r(x)$ and in between $w_{DRF}(x)u$ and $r(x)$.

[9] So, given $w(x)u$ and $w(x)v$, we necessarily have one of these four relations: $w(x)u \rightarrow_H w(x)v$, or $w(x)u \rightarrow_w w(x)v$, or $w(x)v \rightarrow_H w(x)u$, or $w(x)v \rightarrow_w w(x)u$.

Definition: Let $w(x)u$, $r(x)u$ and $w(x)v$ be three operations in $\widehat{H}$ such that $w(x)u \rightarrow_H w(x)v$ or $w(x)u \rightarrow_w w(x)v$. Then we have: $r(x)u \rightarrow_r w(x)v$. (Moreover $\rightarrow_r$ holds only in these cases.)[10]

4.3.2 Acyclicity Lemma

Lemma 4.2 (Acyclicity) *Let $\widehat{H} = (H, \rightarrow_H)$ be a causally consistent history that satisfies the MSC synchronization constraint. Let $\rightarrow_w$ and $\rightarrow_r$ be the two relations on H defined above. Then the relation $\rightarrow_H \cup \rightarrow_w \cup \rightarrow_r$ defines a partial order on H.*

PROOF. Let $\rightarrow$ be either $\rightarrow_H$ or $\rightarrow_w$ or $\rightarrow_r$, and let $\rightarrow_{w/r}$ be either $\rightarrow_w$ or $\rightarrow_r$. Consider the directed graph whose vertices are the operations of H, and whose edges are the $\rightarrow$ relation. We prove that, for any $n > 0$, there are no (directed) cycle of length n in this graph. The proof is by induction on m, where m is the number of $\rightarrow_{w/r}$ edges in the cycle.

i) **Base step** (m=1). Let $op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_n \rightarrow op_1$ be a simple cycle of length $n > 1$, and assume that one single edge of this cycle is of type $\rightarrow_{w/r}$. We show that this leads to a contradiction. Without loss of generality, let $op_1 \rightarrow op_2$ be the only $\rightarrow_{w/r}$ edge in the above cycle; so, as $\widehat{H}$ is transitive, we have $op_2 \rightarrow_H op_1$. There are two cases to consider, numbered $i.1$) and $i.2$).

i.1) $op_1 \rightarrow_r op_2$.
From the definition of $\rightarrow_r$ it follows $op_1 \equiv r(x)u$ and $op_2 \equiv w(x)v$, and either (a) $w(x)u \rightarrow_H w(x)v$ or (b) $w(x)u \rightarrow_w w(x)v$.

(a) $w(x)u \rightarrow_H w(x)v$. As $op_2 \rightarrow_H op_1$, i.e. $w(x)v \rightarrow_H r(x)u$, we have $w(x)u \rightarrow_H w(x)v \rightarrow_H r(x)u$, which means that $r(x)u$ is not legal, in contradiction with the assumption that, because $\widehat{H}$ is a causally consistent history, its read operations are legal.

(b) $w(x)u \rightarrow_w (x)v$. From the definition of $\rightarrow_w$ it follows that $w(x)u$ and $w(x)v$ are concurrent and respectively tagged *DRF* and *CWF*. So we get (1) $w_{CWF}(x)v \rightarrow_H r(x)u$ (because $op_2 \rightarrow_H op_1$), (2) $w_{DRF}(x)u \rightarrow_H r(x)u$ (*read-from* relation), and (3) $w_{CWF}(x)v$ and $w_{DRF}(x)u$ are concurrent. This is in contradiction with the read rule of Section 4.2 (namely $r(x)$ cannot read value u; it reads v or, if it exists, a more recent value v' such that $w(x)v \rightarrow_H w(x)v'$).

[10] Note that in this case u and v are necessarily distinct.

i.2) $op_1 \rightarrow_w op_2$.
By definition of $\rightarrow_w$, op_1 and op_2 are two concurrent write operations, but $op_2 \rightarrow_H op_1$ (because $op_1 \rightarrow op_2$ is the only $\rightarrow_{w/r}$ edge in the cycle): a contradiction.

ii) **Induction step** ($m > 1$). Let $op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_n \rightarrow op_1$ be a simple cycle of length n, and assume that there is no cycle with m or less than m edges $\rightarrow_{w/r}$ ($m < n$). We prove that there can be no cycle with $m+1$ edges $\rightarrow_{w/r}$. The proof is again by contradiction.

Assume a cycle with $m+1$ edges $\rightarrow_{w/r}$, and pick arbitrarily two of these $m+1$ edges. Without loss of generality let op_1, op_2, op_t ($t \geq 2$), op_{t+1} be the four operations that are the endpoints of the two $\rightarrow_{w/r}$ edges: $op_1 \rightarrow_{w/r} op_2$ and $op_t \rightarrow_{w/r} op_{t+1}$. There are four cases to consider, numbered $ii.1$) to $ii.4$).

ii.1) $op_1 \rightarrow_r op_2$ and $op_t \rightarrow_r op_{t+1}$.
By definition of $\rightarrow_r$: op_1, op_t are read operations and op_2, op_{t+1} are write operations.
ii.11). If op_1 and op_2 (or op_t and op_{t+1}) are both tagged *DRF* we have $op_1 \rightarrow_H op_2$ or $op_2 \rightarrow_H op_1$ and consequently there exists a cycle of $\rightarrow_{w/r}$ with less than $m+1$ edges.
ii.12). If both op_1 and op_2 are not tagged *DRF*, and the same holds for op_t and op_{t+1}, then due to the incompatibility of *CRF* and *CWF* tags, either both reads (op_1 and op_t) are tagged *CRF*, or both writes (op_2 and op_{t+1}) are tagged *CWF*. If the reads are tagged *CRF*, then as $\widehat{H}$ satisfies MSC, either $op_1 \rightarrow_H op_t$ or $op_t \rightarrow_H op_1$. If the writes are tagged *CWF*, then as $\widehat{H}$ satisfies MSC, either $op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In all of these four cases, we are able to exhibit a cycle with no more than m edges $\rightarrow_{w/r}$, which is in contradiction with the induction hypothesis: if $op_1 \rightarrow_H op_t$ or $op_{t+1} \rightarrow_H op_2$ (respt. $op_t \rightarrow_H op_1$ or $op_2 \rightarrow_H op_{t+1}$) then there is a cycle not including the edge $op_1 \rightarrow_r op_2$ (respt. $op_t \rightarrow_r op_{t+1}$) and so including less than $m+1$ edges $\rightarrow_{w/r}$.

ii.2) $op_1 \rightarrow_r op_2$ and $op_t \rightarrow_w op_{t+1}$.
By the definition of $\rightarrow_r$ and $\rightarrow_w$, op_1 is a read operation, op_2, op_t, op_{t+1} are write operations and op_{t+1} is tagged *CWF*. Because op_{t+1} is tagged *CWF*, there can be no read operations tagged *CRF*, i.e. op_1 is tagged *DRF* and op_2 is tagged *DRF* or *CWF*.
ii.21). If op_2 is tagged *DRF* then we have $op_1 \rightarrow_H op_2$ or $op_2 \rightarrow_H op_1$ and there is a cycle of $\rightarrow_{w/r}$ of less than $m+1$ edges.
ii.22). If op_2 is tagged *CWF* then, as op_{t+1} is also tagged *CWF*, we have either

$op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In both cases, we can exhibit as previously a cycle with no more than $m \rightarrow_{w/r}$ edges, in contradiction with the induction hypothesis: if $op_2 \rightarrow_H op_{t+1}$ (respt. $op_{t+1} \rightarrow_H op_2$), then there is a cycle not including the edge $op_t \rightarrow_w op_{t+1}$ (respt. $op_1 \rightarrow_r op_2$).

ii.3) $op_1 \rightarrow_w op_2$ and $op_t \rightarrow_r op_{t+1}$.
By renaming op_1 to op_t, op_2 to op_{t+1}, op_t to op_1 and op_{t+1} to op_2, case *ii.3)* becomes identical to *ii.2)*.

ii.4) $op_1 \rightarrow_w op_2$ and $op_t \rightarrow_w op_{t+1}$.
By the definition of $\rightarrow_w$, op_2 and op_{t+1} are write operations tagged CWF, i.e. either $op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In both cases, we can exhibit as previously a cycle with no more than $m \rightarrow_{w/r}$ edges, in contradiction with the induction hypothesis: if $op_2 \rightarrow_H op_{t+1}$ (respt. $op_{t+1} \rightarrow_H op_2$) there is a cycle not including the edge $op_t \rightarrow_w op_{t+1}$ (respt. the edge $op_1 \rightarrow_w op_2$).

□

4.3.3 Proof of the MSC Theorem

Lemma 4.2 has showed that $\rightarrow_H \cup \rightarrow_w \cup \rightarrow_r$ defines a (partial) order on H. The following Lemma 4.3 completes the proof of the MSC theorem by showing a legal sequential history $\widehat{S}$ can be constructed by a topological sort of $(H, \rightarrow_H \cup \rightarrow_w \cup \rightarrow_r)$.

Lemma 4.3 (Legality) *A topological enumeration of* $(H, \rightarrow_H \cup \rightarrow_w \cup \rightarrow_r)$ *produces a sequential history* $\widehat{S}$ *that is legal and equivalent to* $\widehat{H}$.

PROOF. By construction, $\widehat{H}$ and $\widehat{S}$ are defined on the same set of operations, have the same process-order relation and the same read-from relation, so they are equivalent. We have to prove that read operations in $\widehat{S}$ are legal, i.e. that, given any $r(x)u$ in $\widehat{S}$, there is no $op(x)v$, with $v \neq u$, such that: $w(x)u \rightarrow_S op(x)v \rightarrow_S r(x)u$.

The proof of legality of the read operations is by contradiction. Let $\rightarrow_{H/w}$ be either $\rightarrow_H$ or $\rightarrow_w$. Assume there is some $op(x)v$, with $u \neq v$, ordered after $w(x)u$ and before $r(x)u$ in $\widehat{S}$, i.e. $w(x)u \rightarrow_S op(x)v \rightarrow_S r(x)u$. We consider two cases according to $op(x)v$ is a read or a write operation.

i) $op(x)v \equiv w(x)v$.
As all writes on x are totally ordered by $\rightarrow_{H/w}$ and as $\widehat{S}$ is a topological sort

of $(H, \rightarrow_H \cup \rightarrow_w \cup \rightarrow_r)$, if $w(x)u$ is ordered in $\widehat{S}$ before $w(x)v$, then we have $w(x)u \rightarrow_{H/w} w(x)v$. By the definition of $\rightarrow_r$ we have thus $r(x)u \rightarrow_r w(x)v$, i.e. $r(x)u$ is before $op(x)v \equiv w(x)v$ in $\widehat{S}$, which contradicts the assumption, namely $w(x)u \rightarrow_S op(x)v \equiv w(x)v \rightarrow_S r(x)u$.

ii) $op(x)v \equiv r(x)v$.
In this case we have: $w(x)u \rightarrow_S r(x)v \rightarrow_S r(x)u$ and $w(x)v \rightarrow_H r(x)v$ (with $u \neq v$). Two sub-cases have to be considered, as we have either $w(x)v \rightarrow_{H/w} w(x)u$ or $w(x)u \rightarrow_{H/w} w(x)v$.
ii.1) $w(x)v \rightarrow_{H/w} w(x)u$.
In that case we have: $w(x)v \rightarrow_S w(x)u \rightarrow_S r(x)v$. By exchanging u and v, case *ii.1)* becomes case *i)*.
ii.2) $w(x)u \rightarrow_{H/w} w(x)v$.
In that case we have: $w(x)u \rightarrow_S w(x)v \rightarrow_S r(x)u$. It follows this case is similar to case *i)*. □

5 MSC Synchronization and Distributed Systems

Until now no assumption has been made about the implementation of the set X of objects. Specifically this means that the result of the previous Section holds in distributed systems where the objects are possibly replicated (e.g. in order to achieve fault-tolerance). In the sequel we consider replication, and discuss the classical implementation of the DRF, CWF and CRF synchronization constraints in distributed systems. Then we propose a simple protocol to implement MSC synchronization and discuss the practical impact of the MSC synchronization (namely, to face partition failures).

5.1 Implementing DRF, CWF and CRF Synchronization Constraints

Protocols implementing the DRF synchronization constraint have been proposed for a long time in the context of distributed systems where each object has several copies. Actually, these protocols implement atomic consistency which is a consistency criterion stronger than sequential consistency. Simple protocols for CWF and CRF synchronization can be designed; in all these protocols, legality all read operations is guaranteed by the underlying causal memory.

DRF. A simple way to ensure the DRF synchronization consists in using, for each object individually, some form of mutual exclusion. If objects are replicated

this is classically implemented either by a voting protocol [24], or by a more general quorum protocol [13]. Given an object x, let QR_x be a read quorum for x, and QW_x be a write quorum for x. A read or write operation on object x by process P_i, that obey the DRF synchronization, requires for P_i to have the corresponding quorum on x.

CWF or **CRF.** A simple way to ensure the CWF or the CRF synchronization is to use a unique *token*. Let T_{CWF} be the CWF token, and T_{CRF} the CRF token (note that, under the MSC constraint, these tokens cannot co-exist). The T_{CWF} token (respt. T_{CRF}) gives its current owner a universal right to write (respt. read) all the objects. More precisely, a write operation on object x by process P_i, that obey the CWF synchronization, requires for P_i to have the T_{CWF} token. Similarly a read operation on object x by process P_i, that obey the CRF synchronization, requires for P_i to have the T_{CRF} token[11].

5.2 An Implementation of the MSC Synchronization Constraint

The advantage of the MSC synchronization over the pure DRF synchronization, or the pure CWF synchronization, is to give two chances to perform an operation: the MSC_W synchronization gives two chances to perform a write operation, whereas the MSC_R synchronization gives two chances to perform a read operation:

MSC_W. To write an object x, a process must either have the write quorum QW_x, or must have the T_{CWF} token. To read an object x, a process must have the read quorum QR_x.

MSC_R. To read an object x, a process must either have the read quorum QR_x, or must have the T_{CRF} token. To write an object x, a process must have the write quorum QW_x.

Consider the protocol implementing MSC_W. As there are two chances to perform a write operation, if the process willing to write an object x requests both the write quorum QW_x and the token T_{CWF}, it will be allowed to write x as soon as one of the two conditions is fulfilled. This shows that the MSC constraint is less constraining than either DRF, CWF or CRF considered alone.

Albeit tokens T_{CWF} and T_{CRF} cannot exist simultaneously, the MSC synchronization has a nice property. It is possible, during an execution, to switch from the

[11] A token can be made fault-tolerant by replication. In this case, to *have the token* is equivalent to have a write quorum on the replicated object representing the token.

MSC_W to the MSC_R synchronization, and from the MSC_R synchronization to the MSC_W synchronization. In other words, the T_{CWF} token can be dynamically changed to a T_{CRF} token, and conversely. The switching condition is the following: a process P_i can change the attribute of the token if and only if (1) P_i owns the token, and (2) P_i has a read quorum QR_x on all the objects of X. (Note that when this condition is fulfilled, no process P_j can concurrently write any object.)

5.3 Network Partitions

The fact that MSC_W gives two chances to perform a write operation, and MSC_R two chances to perform a read operation, is particularly interesting in the case of network partitions. Consider for example the network partitioned into Π_1 and Π_2. The MSC_R synchronization might allow to read an object x in partition Π_1 even if there is no read quorum for x in Π_1 (assume T_{CRF} in Π_1). Similarly the MSC_W synchronization might allow to write an object x both in partition Π_1 and in partition Π_2 (assume T_{CWF} in Π_1, and a write quorum for x in Π_2). One could also imagine the case in which no partition has the write quorum on an object x; despite this, the MSC_W synchronization might allow to read and write x in some partitions (assume the read quorum for x in Π_1 and the T_{CWF} token in Π_2).

Thus the MSC synchronization, while ensuring sequential consistency, is flexible and reduces the blocking period of processes.

6 About Other Consistency Criteria

As indicated in the introduction, this paper aimed at showing that sequential consistency can be obtained out of causal consistency by adding appropriate and flexible synchronization constraints. Some authors have defined other consistency criteria to get efficient parallel programs. They also have given conditions to obtain sequential consistency out of these consistency criteria. We present some of them in the next subsections. Relationships between PRAM consistency and causal consistency are first given. Hybrid consistency, mixed consistency, entry consistency and release consistency are then discussed. A short comparison between the message passing model and the shared memory model concludes this Section.

6.1 PRAM Consistency

PRAM (Pipelined RAM) consistency [18] is a consistency criterion weaker than causal consistency. The difference, in the shared memory model, between PRAM

consistency and causal consistency is the same as the one between FIFO ordering and causal ordering for message deliveries in the message passing model [10, 22]. PRAM and FIFO are only concerned by "direct relations" between pairs of "adjacents" processes and do not take into account transitivity due to intermediary processes. More precisely, in a message passing system with FIFO ordering, two messages sent to a same process by two distinct senders can be delivered in any order, even if the send events are causally related [16] (this is not the case with causal ordering: if the send events are causally related, messages must be delivered in their sending order to the destination process). In the same way, in a PRAM consistent shared memory system, two updates of objects by two distinct processes can be know in any order by a third one[12] (this is not the case in a causally consistent shared memory : if $w(x)u \rightarrow_H w(x)v$, a process reading x can never get v and then u). As an example consider history $\widehat{H_3}$ which is not causally consistent (Figure 3). This history is PRAM consistent: as $w_1(x)1$ and $w_2(x)2$ have been issued by distinct processes, values 1 and 2 of x can be known by P_3 in any order.

Let $\widehat{H}$ be a history and let $\widehat{H'} = (H', \rightarrow_{H'})$ be a history defined from $\widehat{H}$ in the following way ($\widehat{H'}$ differs from $\widehat{H}$ only in point *iii)* defining transitivity, where $\rightarrow_i$ is used instead of $\rightarrow_H$):

- $H' = H$ (so $H' = \bigcup_i h_i$)
- $op1 \rightarrow_{H'} op2$ if :

  i) $\exists\ P_i$: $op1 \rightarrow_i op2$ (*process-order* relation),

  or *ii)* $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (*read-form* relation),

  or *iii)* $\exists op3$: $op1 \rightarrow_i op3$ and $op3 \rightarrow_i op2$.

Let $\widehat{H_i}$ be the sub-history of $\widehat{H'}$ from which all read operations not issued by P_i have been removed. $\widehat{H}$ is PRAM consistent if, for each P_i, there exists a legal sequential history $\widehat{S_i}$ which is equivalent to $\widehat{H_i}$. This definition of PRAM consistency shows that its difference, with respect to causal consistency, lies in the nature of the transitivity considered.

6.2 Mixed Consistency

Mixed consistency has been introduced by Agrawal *et al.* in [2]. This consistency criterion on one side considers histories including memory operations (read and

[12] Of course, only one of the updates can be know, if updates overwrite the value previously written by other processes.

write) and synchronization operations (lock, barrier and await), and on the other side combines PRAM consistency with causal consistency; namely every read operation is tagged either *PRAM* or *causal*.

A history $\widehat{H}$ is mixed consistent if it is:

- causally consistent when considering only the legality of read operations tagged *causal*, and
- PRAM consistent when considering only the legality of read operations tagged *PRAM*.

The following result is shown in [2]. A mixed consistent history $\widehat{H}$ in which all read operations are tagged *causal*[13] and in which every pair of concurrent operations commute[14], is sequentially consistent.

6.3 Hybrid Consistency

Hybrid consistency has been introduced by Attiya and Friedman in [6]. This consistency criterion guarantees properties on the order in which operations appear to be executed at the program level. Operations are labeled either *strong* or *weak*. By defining which operations are strong and which are weak, a user can tune the consistency criterion to his own need. Informally hybrid consistency guarantees the following two properties:

- all strong operations appear to be executed in some sequential order,
- if two operations are invoked by the same process and at least one of them is strong, then they appear to be executed in their invocation order to all processes.

Hence all processes agree on a total order for all strong operations, and on the same order for any pair of strong and weak operations issued by the same process. They can disagree on the relative order of any pair of weak operations issued by a process between two strong operations. The following result is proved in [7]:

- every hybrid consistent history in which all writes are strong and all reads are weak is sequentially consistent;

[13] Note $\widehat{H}$ is then causally consistent.

[14] Two concurrent operations commute if their execution order is irrelevant (this is not the case for two concurrent writes on a same object).

- every hybrid consistent history in which all writes are weak and all reads are strong is sequentially consistent.

This result (a hybrid history that satisfies the previous property is sequentially consistent) is similar to the one implied by the synchronization constraint CWF or CRF. But it requires more synchronization than the MSC synchronization constraint, as it does not accept either concurrent write operations or concurrent read operations. The MSC constraint potentially allows more parallelism to get sequential consistency (this results from the two layers approach with a basic layer providing an underlying causally consistent memory).

6.4 Non Primitive Read and Write Operations

Till now we have supposed that read and write operations offered to users are primitive operations. Some authors have considered to provide users with mechanisms allowing them to define non primitive read and write operations on a set of shared data objects (notation: READ, WRITE). Each such READ or WRITE operation is actually a procedure bracketed by two synchronization operations (*release* and *acquire*), and composed of non synchronized primitive read and write operations. Both *release* consistency [12] and *entry* consistency [9] address such non primitive READ and WRITE operations, and provide sequential consistency when acquire and release operations guarantee the readers-writers discipline. Concerning protocols implementing these consistency criteria, eager *vs* lazy [15] is an implementation issue whose aim is to reduce the number of messages and the amount of data exchanged; invalidation *vs* update is another implementation issue addressing the management of multiple copies of objects when a cached-based approach is used.

6.5 Shared Memory Model *vs* Message Passing Model

The definition of an history in the shared memory model, and its definition in the message passing model, have some similarities. The first definition of an history in the message passing model is due to Lamport [16]. In the message passing model, operations issued by a process are modeled as events and can be :

- the sending of a message m (event $send(m)$);
- the reception of a message m (event $receive(m)$);
- the execution of a statement involving neither the send nor the receive of a message (*internal* event).

The local history of a process P_i is the sequence $\widehat{h_i}$ of events it has produced. A distributed computation (or an history) $\widehat{H}$ of a set of processes $P_1, ..., P_n$ is a partial order $(H, \rightarrow_H)$ defined as in the case of the shared memory model except for point *ii)*. More precisely, the definition of the *read-from* relation is replaced by the definition of a *message* relation :

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ if :

  i) $\exists\ P_i\ : op1 \rightarrow_i op2$ (in that case, $\rightarrow_H$ is called *process-order* relation),

  or *ii)* $op1 = send(m)$ and $op2 = receive(m)$ (in that case $\rightarrow_H$ is called *message* relation),

  or *iii)* $\exists op3\ : op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$.

The similarity between both models is not limited to their definitions. PRAM, causal and sequential consistencies in the shared memory model correspond to FIFO, causal and logically instantaneous [23] communications in the message passing model. Characterizations of these communication modes can be found in [8, 11, 23]. Let $\widehat{H}$ be a history in the message passing model. In [11], the following results are shown:

- $\widehat{H}$ has causally ordered communications if it satisfies the empty interval property, namely:

$$\forall\ receive(m) \in \widehat{H} : \{op \mid send(m) \rightarrow_H op \rightarrow_H receive(m)\} = \emptyset \qquad (1)$$

- $\widehat{H}$ has logically instantaneous communications if it has a linear extension satisfying the empty interval property. In others words it must exist a sequential history $\widehat{S}$ (1: linear extension) which is equivalent to $\widehat{H}$ (same operations, same process-order and message relations) and in which (2: empty interval property) for all messages m, the receive event follows immediately the corresponding send event:

$$\forall\ receive(m) \in \widehat{S} : \{op \mid send(m) \rightarrow_S op \rightarrow_S receive(m)\} = \emptyset \qquad (2)$$

These characterizations of causal and logically instantaneous communication modes are very similar to their counterpart in the shared memory model, causal consistency and sequential consistency respectively. Let $\widehat{H}$ be a history in the shared memory model. Causal consistency and sequential consistency can be described in the following way:

- $\widehat{H}$ is causally consistent if all its read operations are legal, i.e. :

$$\forall\ r(x)v \in \widehat{H} : \{op(x)u \mid u \neq v \wedge w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v\} = \emptyset \qquad (3)$$

- $\widehat{H}$ is sequentially consistent if it exists an equivalent sequential history $\widehat{S}$ in which all read operations are legal, i.e. :

$$\forall\ r(x)v \in \widehat{S} : \{op(x)u \mid u \neq v \wedge w(x)v \rightarrow_S op(x)u \rightarrow_S r(x)v\} = \emptyset \qquad (4)$$

Similarities between (1) and (3) on one side, and (2) and (4) on the other side, are evident.

7 Conclusion

This paper has studied synchronization constraints that, when obeyed by operations of a given causally consistent execution, make it sequentially consistent. Such an approach is particularly interesting as, from methodological and implementation points of view, it means that a protocol implementing sequential consistency can consist of two independent layers: a basic one implementing causal consistency and, on top of it, another one implementing some synchronization constraints for the operations issued by processes.
The paper introduced the MSC synchronization (mixed synchronization constraint) which generalizes (1) the known DRF (data race free) and CWF (concurrent write free) synchronizations and (2) a new one called CRF (concurrent read free). A main interest of this constraint lies in the fact it allows concurrent conflicting operations on a same object while ensuring sequential consistency; this is particularly interesting in the the context of distributed systems (where objects are possibly replicated) to cope with partition failures: conflicting operations in two distinct partitions do not necessarily block processes that issue them (as it is the case with quorum based protocols). Technically, a tag (control type) is associated with each operation, and all operations endowed with the same tag obey the same constraint.
The paper has also classified the synchronization constraints in two classes: the *per object* synchronization class, and the *inter-object* synchronization class (which includes the per object synchronization class). This classification allows to better understand linearizability (which has the nice locality property [14]) with respect to sequential consistency: linearizability is obtained by the *per object* synchronization.
Finally, while the paper has identified MSC as a sufficient condition to get sequential consistency out of causal consistency, it would be interesting to identify a necessary condition to get sequential consistency out of causal consistency.

Aknowledgment

This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by France Telecom under a CNET grant "Cohérence d'objets répartis", and by the "Fonds national suisse" and OFES under contract number 21-32210.91.

References

[1] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Prog. Lang. and Systems*, 15(1):182–205, 1993.

[2] D. Agrawal, M. Choy, H.V. Leong and A. Singh. Mixed consistency: a model for parallel programming. *In Proc. 13th ACM Symposium on Principles of Dist. Computing*, Los Angeles, pages 101–110, 1994.

[3] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neiger. Causal Memory. In *Proc. 5th Intl. Workshop on Distributed Algorithms (WDAG-5)*, pages 9–30. Springer Verlag, LNCS 579, 1991.

[4] M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. TR GIT-CC-93/55, Georgia Institute of Technology, July 94, 25p.

[5] H. Attiya and J.L. Welch. Sequential Consistency versus Linearizability. *ACM Trans. on Comp. Systems*, 12(2):91–122, 1994.

[6] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. *In proc. 24th ACM Annual Symposium on the Theory of Computing*, pages 679–690, 1992.

[7] H. Attiya, S. Chaudhuri, Friedman R. and J.L. Welch. Shared memory consistency conditions for non sequential executions: definitions and programming strategies. *In proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, Vale, Germany, july 1993.

[8] R. Baldoni and M.Raynal. A graph-based characterization of communications modes in distributed executions. *Journal of Foundations of Computing and Decision Sciences*, 25(1), 1995.

[9] B.N. Bershad, M.J. Zekauskas and W.A. Sawdon. The Midway distributed shared memory system. *Proc. of the Compcon 93 Conference*, pages 528–537, Feb. 1993.

[10] K. Birman and T. Joseph. Reliable communications in the presence of failures. *ACM Trans. on Comp. Systems*, 5(1):47-76, 1987.

[11] B. Charron-Bost, F. Mattern and G. Tel. Synchronous and asynchronous communications in distributed systems. Tech. Report TR91.55, University of Paris 7, September 1991.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessey. Memory consistency and event ordering in scalable shared memory multiprocessors. *Proc. 17th Annual Int. Symposium on Comp. Architecture*, Seattle, WA, pages 15–26, 1990.

[13] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed systems. *Journal of the ACM*, 32(4):841–850, 1985.

[14] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Systems*, 12(3):463–492, 1990.

[15] P. Keleher, A.L. Cox and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Computer Architecture News*, 22(2):13–21, 1992.

[16] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C28(9):690–691, 1979.

[18] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Tech. Report CS-TR-180-88, Princeton University, Sept. 1988.

[19] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Prog. Lang. and Systems*, 8(1):142–153, 1986.

[20] M. Mizuno, M. Raynal, and J.Z. Zhou. Sequential Consistency in Distributed Systems: Theory and Implementation. *Proc. Int. Workshop "Unifying Theory and Practice in Dist. Systems"*, Dagstuhl, Germany, to appear in LNCS Series, Springer-verlag (K.Birman, F. Cristian, F. Mattern and A. Schiper Eds), (also TR 871, Irisa, Rennes, October 1994, 41p.).

[21] D. Mosberger. Memory consitency models. *ACM Operating Systems Review*, 27(1):18–26, 1993.

[22] M. Raynal, A. Schiper and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Proc. Letters*, 39:343–350, 1991.

[23] T.S. Soneoka and T. Ibaraki. Logically instantaneous message passing in asynchronous distributed systems. *IEEE Trans. on Computers*, 43(5);513–527, 1994.

[24] R.H. Thomas. A majority consensus approach to concurrency control for multiple copies databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399